behaviac 概述

腾讯游戏
Tencent Games

https://github.com/TencentOpen/behaviac

目录

1 项目概况

1.1 behaviac 组件

- behaviac 是游戏 AI 的开发框架组件，也是游戏原型的快速设计工具
- 支持行为树 BT，状态机 FSM，HTN 等多种范式，方便的编辑和调试
- 支持全平台，适用于客户端和服务器，助力游戏快速迭代开发
- 在公司内部已经有多个项目在使用，包括《天天炫斗》、《QQ 飞车》、《全民夺宝》、《全面突击》、《九龙战》以及其他若干预研项目，并已在 GitHub、CSDN 等国内外知名网站进行开源
- 支持 C++和 C#两种编程语言，并支持所有的主流平台，包括 Windows/Linux/Android/iOS 等，对 Unity 引擎有 C#的原生支持
- 功能强大，扩展性较强，文档丰富，支持中英文界面
- 该组件的使用场景，支持但不限于游戏中的逻辑、角色的人工智能、动画的控制等方面

1.2 特性及用户价值

- 编辑器和 C++/C#运行时的交互基于元信息，既能充分利用程序代码的各种功能，也提供了对高层逻辑的图形化控制
- 在编辑器中就可以创建和编辑元信息，方便策划提前实现原型
- 支持 XML、BSON、C++、C#等格式的文件导出，既提供了加载、执行的高效，也使用热加载极大的提高了开发效率
- 功能完善易用的编辑器，支持 Prefab、Undo/Redo、子树、事件等
- 支持的数据类型和节点类型可方便的扩展
- 实时或者离线调试，将执行逻辑图形化显示，方便调试
- 使用场景，不只是 AI，支持并不限于：
  - ✧ Character AI
  - ✧ Squad Logic
  - ✧ Strategy AI
  - ✧ In-Game Tutor
  - ✧ Animation Control
  - ✧ Player Avatars

1.3 项目链接

可以参考后面网址获取最新版本等更多信息：https://github.com/TencentOpen/behaviac

2 背景

游戏 AI 的目标之一就是要找到一种简单并可扩展的开发逻辑的方案，常用的技术包括有限状态机（FSM）、分层有限状态机（HFSM）、面向目标的动作规划（GOAP）、分层任务网络（HTN）等。

行为树作为次时代的 AI 技术，距其原型提出也约有 10 年左右，像 Halo、Spore、Crysis 等大作已经采用。目前，很多知名的游戏引擎也已整合或提供了自己的行为树组件，例如 Unreal4、Unity 引擎等。

2.1 有限状态机

提到有限状态机（Finite State Machine，FSM），很多程序员都相当熟悉。状态机技术在游戏开发中也已变得很成熟和流行，它反映了从系统开始到现在的输入变化。

- 动作（Action）：指在给定时刻要进行的活动的描述。
- 状态（State）：指对象的某种形态，在当前形态下可能会拥有不同的行为和属性。
- 转换（Transition）：表示状态变更，并且必须满足确使转移发生的条件来执行。
- 状态机（State Machine）：控制对象状态的管理器。对象的状态不会无端的改变，它需要在某种条件下才会变换。状态会在某个事件触发之后变更，不同的状态也有可能决定了对象的不同属性和行为。

如图 2.1 所示，有限状态机维护了一张图，图的节点是一个个的状态，节点和节点的连线是状态间根据一定规则所执行的状态转换，每一个状态内的逻辑都可以简要描述为：

如果满足条件 1，则跳转到状态 1；

如果满足条件 2，则跳转到状态 2；

…；

否则，不做任何跳转，维持当前状态。

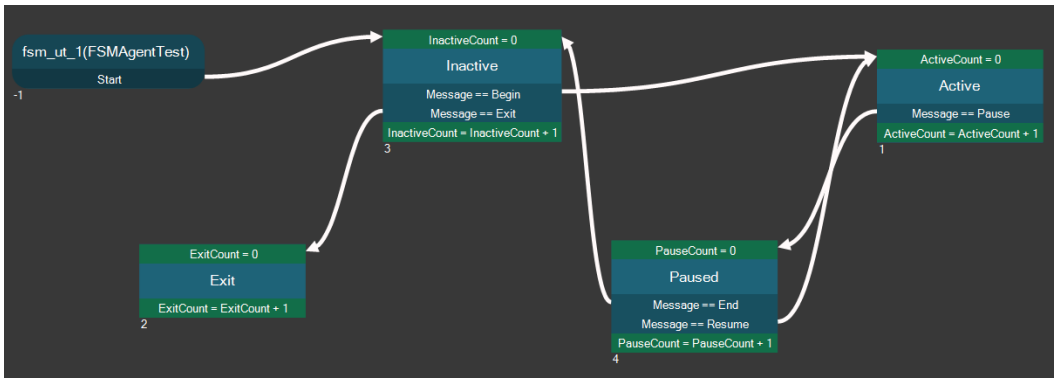

图 2.1 有限状态机

有限状态机有着简单的优势，采用状态机来实现 AI 更符合思维的朴素表达。对于一些简单的 AI，用状态机更加便捷，但是面对一些复杂的 AI 逻辑就会显得比较繁杂。同时，当需要对现有行为逻辑进行扩展的时候，代码上会显得比较吃力，因为要维护的状态量会成倍

增加。

对于大型的系统，分层有限状态机（HFSM）也支持状态间切换的重用。但是状态机需要用转换连接状态，从而状态失去了模块性。

2.2 行为树

行为树（Behavior Tree，BT），使得实现 AI 的过程变得更加需要技巧。框架设计者较为全面的考虑了可能会遇到的种种情况，把每种情况都抽象成了某个类型的节点，而游戏开发者要做的就是按照规范把各种节点连接成一棵所需的行为树。这样，行为树更加具有面向对象的特征，行为模块间的藕合度相对较低。

概念上，行为树就是一段脚本，以树的形式展现给用户。节点的执行结果由其父节点来管理，决定接下来做什么。由于节点间不再有转换，因此不再称为状态，节点只是行为。

行为树的基本结构如图 2.2 所示，行为树由叶子节点和中间节点组成，从左到右依次是父子关系的节点：叶子节点主要是一些动作、条件和赋值等原子操作节点，包含了最基本的行为（如跑动、攻击等），当一个叶子节点被选择后，就会激活其对应的基本行为；中间节点主要是一些组合节点，代表逻辑单元，用于管理子节点如何执行等。

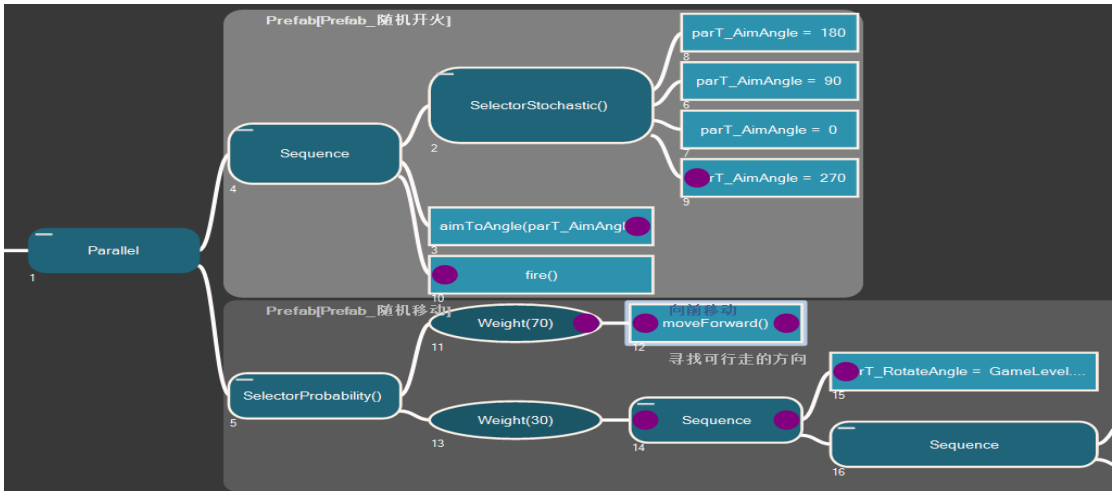

图 2.2 行为树

叶子节点和中间节点主要分为五大类节点或附件：动作、条件、组合、修饰、附件等。其中，动作、条件节点为叶子节点，组合、修饰节点为中间节点，附件必须附属在这四类节点上面而不能独立存在。

每个节点执行结束后，都必须将其返回值提供给父节点。节点的返回值包括三种：成功（Success）、失败（Failure）以及正在执行（Running）。最基本的行为（叶子节点）可能执行成功也可能失败，高等级的行为（中间节点）是否执行成功依赖于子节点是否执行成功，子节点执行失败可能导致其父节点选择另一个子节点。

行为树中最常用的节点包括：

- 动作节点（Action）：属于叶子节点，用于描述一个最终执行的动作。
- 条件节点（Condition）：属于叶子节点，用于描述一个条件是否成立。
- 选择节点（Selector）：属于组合节点，用于顺序执行子节点，只要它的一个子节点返回成功，则整个分支返回成功，反之返回失败，类似程序中的逻辑或（OR）。
- 顺序节点（Sequence）：属于组合节点，用于顺序执行子节点，只要它的一个子节

点返回失败，则整个分支返回失败，反之返回成功，类似程序中的逻辑与（AND）。

行为树的执行通过帧循环的更新来驱动，不一定是每帧都需要更新，但是需要周期性的执行。基于效率的考虑，行为树的执行有点类似于协程（Coroutine）的概念。如果有返回正在执行的节点，那么在行为树下一次执行的时候会接着执行。否则，一个返回成功或失败也就是已经结束了的行为树，下一次执行就会从根节点重新开始。

一棵行为树首先需要设置一个 Agent 类型，Agent 也就是游戏中的 AI 角色。在这棵行为树中的所有节点（主要是叶子节点），可以进一步选择 Agent 的属性、方法以及其他变量等进行配置。这些 Agent 的类型、属性、方法等信息，称之为元信息，后文会更详细的对其进行介绍。

行为树的优势如下：

- 行为逻辑和状态数据分离，任何节点都可以反复利用。
- 重用性高，可用通过重组不同的节点来实现不同的行为树。
- 呈线性的方式布局，易扩展。
- 可配置，把工作交给策划。

3 behaviac 组件

behaviac 组件包括编辑器（Designer）和运行时（Runtime）两大部分：编辑器主要用于编辑行为树，运行时库主要用于解释和执行编辑并导出过的行为树，运行时库需要整合到自己的游戏项目中去。

编辑器和运行时之间通过元信息（Meta）进行交互，如图 3.1 所示：

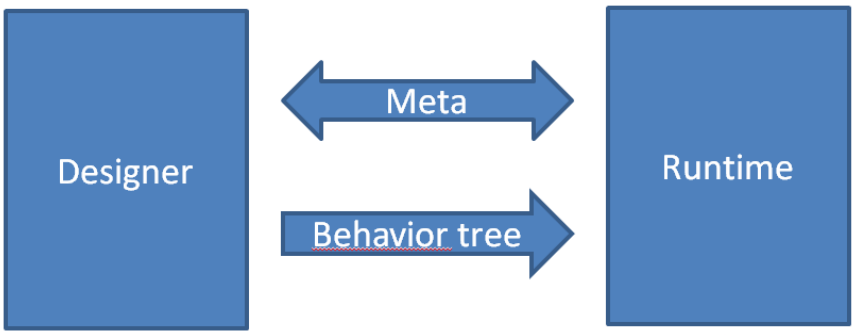

图 3.1 基于元信息的编辑器和运行时端的交互

3.1 元信息

元信息是 behaviac 组件的核心，包括 Agent 的类型、属性、方法以及类的实例名称等信息。游戏中负责运行行为的实体称谓 Agent，每个 Agent 都存有自己的数据或者能执行自己的动作，这样的数据被称为属性（Property），这样的动作被称为方法（Method）。所谓的元信息就是对一个 Agent 的描述，包括其属性和方法等。

运行时库产生一个描述 Agent 元信息的 XML 文件。注意图 3.1 中元信息的双向箭头，“双向”表示运行时端可以导出元信息给编辑器使用，在编辑器中也可以编辑并导出元信息到运行时端。

在运行时端，也即游戏代码端，通过注册（C++通过宏的方式，C#通过标记 Attribute 的方式，如图 3.1.1 和 3.1.2 所示）并导出供行为树引擎和编辑器使用的 XML 元信息。运行时端主要是由程序员编写 Agent 子类及其属性和方法，然后调用相关接口将这些元信息导出，就可以在编辑器中对这些元信息进行使用。

```cpp
BEGIN_PROPERTIES_DESCRIPTION(AgentNodeTest)
{
    //CLASS_DISPLAYNAME(L"测试behaviac::Agent")
    //CLASS_DESC(L"测试behaviac::Agent的说明")
    REGISTER_PROPERTY(testVar_0);
    REGISTER_PROPERTY(testVar_1).DISPLAYNAME(L"testVar_1").DESC(L"testVar_1 property").RANGE(100);
    REGISTER_PROPERTY(testVar_2);
    REGISTER_PROPERTY(testVar_3);
    REGISTER_PROPERTY(waiting_timeout_interval);
    REGISTER_PROPERTY(testVar_str_0);

    REGISTER_METHOD(setEventVarInt);
    REGISTER_METHOD(setEventVarBool);
    REGISTER_METHOD(setEventVarFloat);
    REGISTER_METHOD(getConstOne);
    REGISTER_METHOD(setTestVar_0);
    REGISTER_METHOD(setTestVar_1);
    REGISTER_METHOD(setTestVar_2);
    REGISTER_METHOD(setTestVar_0_2);
    REGISTER_METHOD(setTestVar_R);
    REGISTER_METHOD(setTestVar_3);
    REGISTER_METHOD(enter_action_0);
    REGISTER_METHOD(exit_action_0);
    REGISTER_METHOD(enter_action_1);
    REGISTER_METHOD(exit_action_1);
    REGISTER_METHOD(enter_action_2);
    REGISTER_METHOD(exit_action_2);
    REGISTER_METHOD(createGameObject);
    REGISTER_METHOD(testGameObject);
    REGISTER_METHOD(switchRef);
}
END_PROPERTIES_DESCRIPTION()
```

图 3.1.1 在 C++中通过宏注册元信息

```cpp
[behaviac.TypeMetaInfo()]
342 references
public class AgentNodeTest : behaviac.Agent
{
    [behaviac.MemberMetaInfo()]
    public int testVar_0 = -1;

    [behaviac.MemberMetaInfo("testVar_1", "testVar_1 property", 100)]
    public int testVar_1 = -1;

    [behaviac.MemberMetaInfo()]
    public float testVar_2 = -1.0f;

    [behaviac.MemberMetaInfo()]
    public float testVar_3 = -1.0f;

    [behaviac.MemberMetaInfo()]
    public int waiting_timeout_interval = 0;

    [behaviac.MemberMetaInfo()]
    public string testVar_str_0 = string.Empty;

    [behaviac.MethodMetaInfo()]
    3 references
    public void setEventVarInt(int var) {
        event_test_var_int = var;
    }

    [behaviac.MethodMetaInfo()]
    3 references
    public void setEventVarBool(bool var) {
        event_test_var_bool = var;
    }

    [behaviac.MethodMetaInfo()]
    1 reference
    public void setEventVarFloat(float var) {
        event_test_var_float = var;
    }
```

图 3.1.2 在 C#中通过 Attribute 标记元信息

此外，behaviac 组件的特色之一就是编辑器中也可以创建和编辑元信息。编辑器主要交给策划编辑行为树，也可以编辑元信息。策划在项目开始初期，也就是程序员还没把代码写出来之前，策划就可以自己手动的创建一些 Agent 类型、属性和方法等元信息。这样可以加速游戏原型的创建，也就是策划不用等程序员，就可以进行游戏原型的编辑。

如图 3.1.3 所示，打开编辑器中提供的元信息浏览器，可以查看或新建 Agent 类型。选中其中一个 Agent 类型之后，可以查看它的属性、方法等信息，并可以扩展 Agent 类型的成员属性和方法等，还支持新建枚举和结构体类型等。

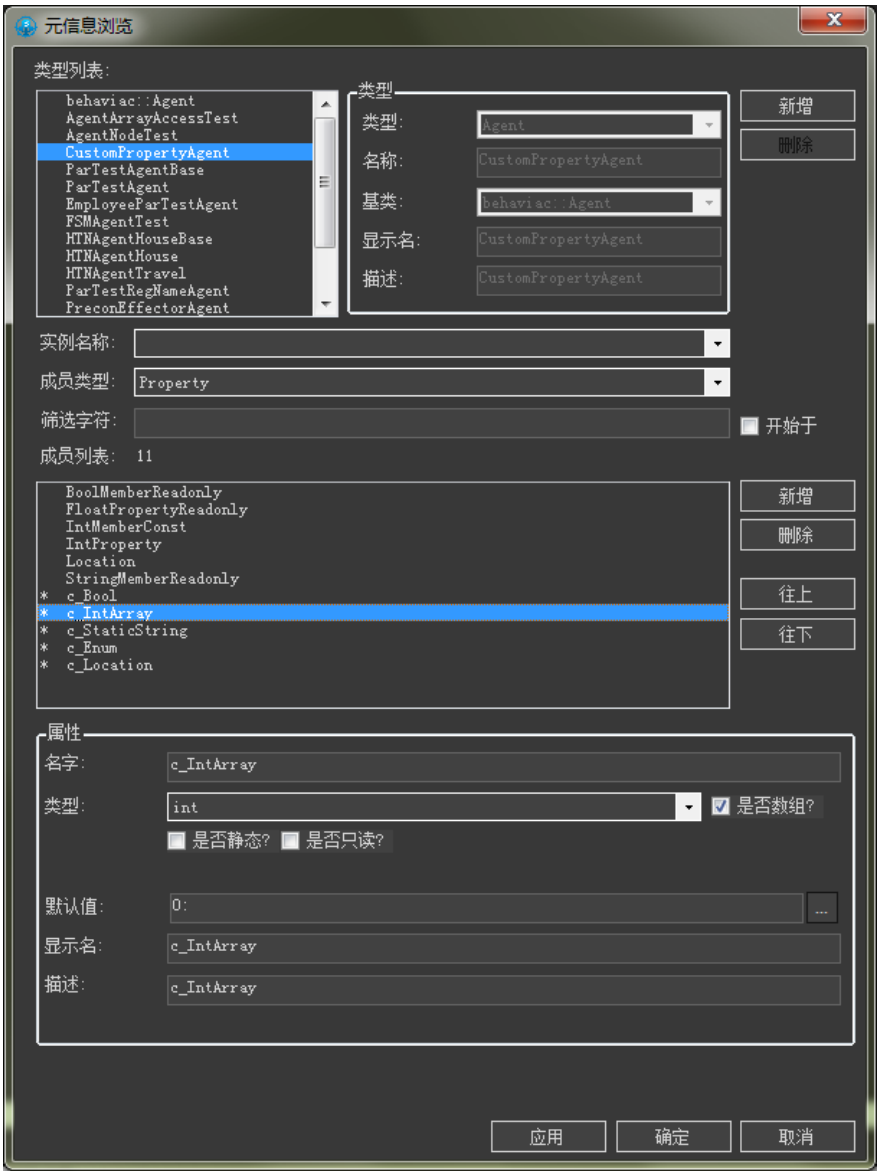

图 3.1.3 在元信息浏览器中查看和编辑元信息

3.1.1 类型

类型（Type）分为 Agent、枚举和结构体等。

- **Agent 类：**作为行为树的基本组成元素，是供其他游戏类派生的基类，在图 3.1.3 所示的元信息浏览器中，我们可以查看所有导出和创建的 Agent 类型，这些 Agent 类型又包含变量和方法两大部分。可以在元信息浏览器中添加自己的 Agent 子类，

为其添加成员属性和方法，但方法的实现需要另外编写需要的逻辑代码。

- **枚举**：游戏端的用到枚举类型会导出在元信息文件中，也可以在元信息浏览器中添加自己的枚举类型，并可以导出生成 C++或 C#源码，再添加到游戏端代码中。
- **结构体**：游戏端的用到结构体类型会导出在元信息文件中，也可以在元信息浏览器中添加自己的结构体类型，但只能为其增加成员属性，不能添加成员方法，可以导出生成 C++或 C#源码，添加到游戏端代码中。

如图 3.1.1.1 所示，在新建一棵行为树时，首先需要为根节点选择一种 Agent 类型，以表明当前新建的行为树是用的哪种 Agent，该行为树的所有叶子节点将可以使用该 Agent 类型的属性和方法。

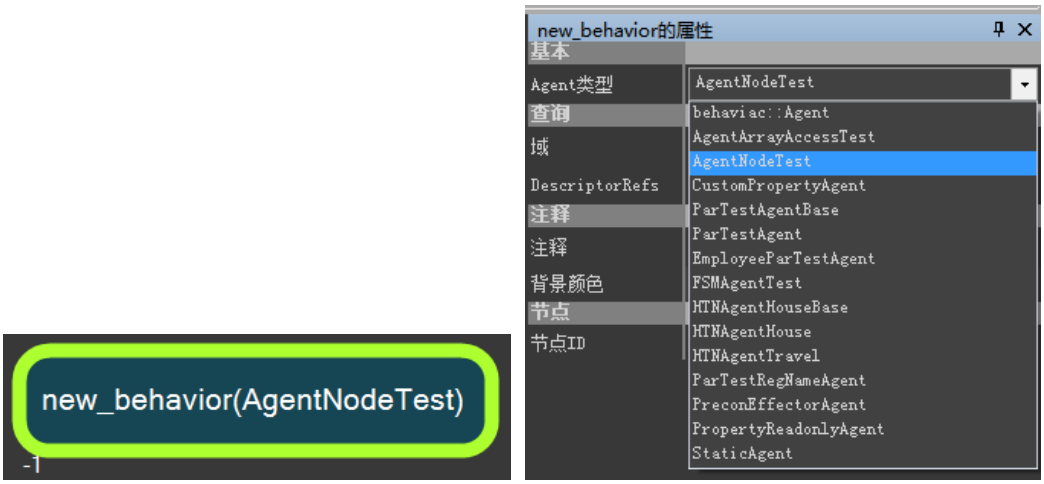

图 3.1.1.1 为根节点设置 Agent 类型

3.1.2 实例

上面已经提及了一棵行为树需要设置它的 Agent 类型，以便于所有叶子节点使用该 Agent 类型的属性和方法。对于这种情况，我们称之为使用自身（Self）的属性和方法。

另一方面，某个节点的参数可能会用到不是自身的属性或方法，这时就需要首先选择实例（Instance），然后再选择该实例所属的 Agent 类型的属性或方法，如图 3.1.2.1 所示。

所有 Agent 类的实例在运行时端需要通过注册并导出到元信息文件中，这样编辑器拿到所有的实例列表后就可以一一列举出来以供选择，具体注册方法详见使用手册。

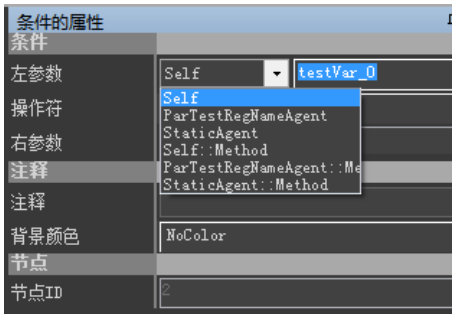

图 3.1.2.1 为条件节点设置左参数

3.1.3 变量

变量分为成员字段（Field）、属性（Property）、定制的属性以及局部变量等。

- **成员字段**：也即面向对象编程（C++/C#）中类的字段。
- **成员属性**：也即 C#中类的属性，带有 getter 和 setter。
- **定制的属性**：主要是在编辑器中为 Agent 类型新建出来的成员属性。
- **局部变量**：相对于成员属性和定制的属性这些全局属性而言。为什么称之为全局属性？在整个游戏项目中，可能会有很多棵行为树，我们称之为工作区来管理所有的行为树。每一种 Agent 类型的成员属性和定制的属性是可以供这个工作区中的所有行为树选择使用的，而局部变量只是针对某一棵行为树，局限于在这一棵行为树中所使用的变量。

behaviac 组件支持变量的只读（Readonly）特性，这样在编辑器中只可以对该属性进行读取，但不能为其赋值。局部变量不仅支持 int、float、bool、枚举等基本的数据类型，也支持复杂的指针或引用类型。

3.1.4 方法

方法包括成员方法、定制的方法、任务（Task）等。

- **成员方法**：也即面向对象编程中类的成员方法。
- **定制的方法**：在编辑器中创建出来的方法。
- **任务**：用于描述子树所需的接口，包括接口名和参数列表，跟方法的定义类似。后文将结合子树的概念，对任务的用法进行更详细的说明。

最后，在编辑器中就可以根据导出和创建的元信息，对行为树进行编辑。在行为树的每个节点上可以根据需要选择或设置属性、方法及其参数等，如图 3.1.4.1 所示。

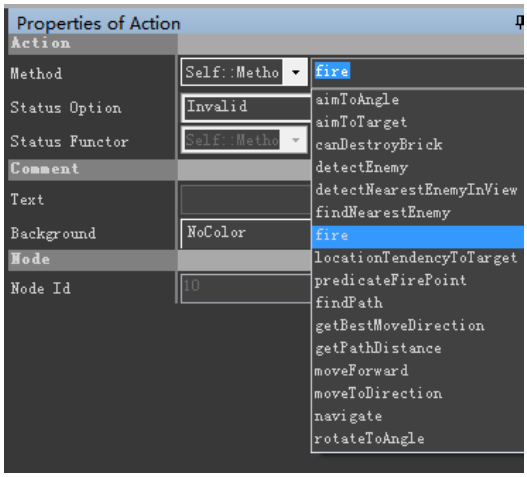

图 3.1.4.1 编辑节点的属性

3.2 工作区

采用 behaviac 组件开始项目时，首先需要在编辑器中创建一个工作区（Workspace）。

工作区文件以.workspace.xml 为后缀名，是管理游戏项目中所有行为树文件的配置文件，如图 3.2.1 所示：

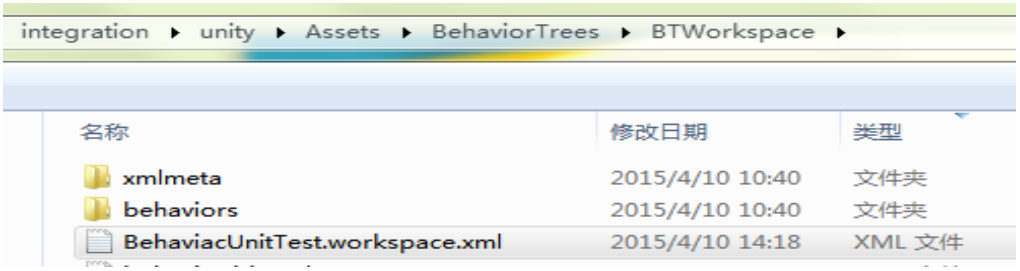

图 3.2.1 工作区文件

该配置文件可以指定 XML 元信息文件、行为树源文件的路径、导出文件的路径等，可以通过编辑器中的“新建工作区”或“编辑工作区”工具编辑相关设置，如图 3.2.2 所示：

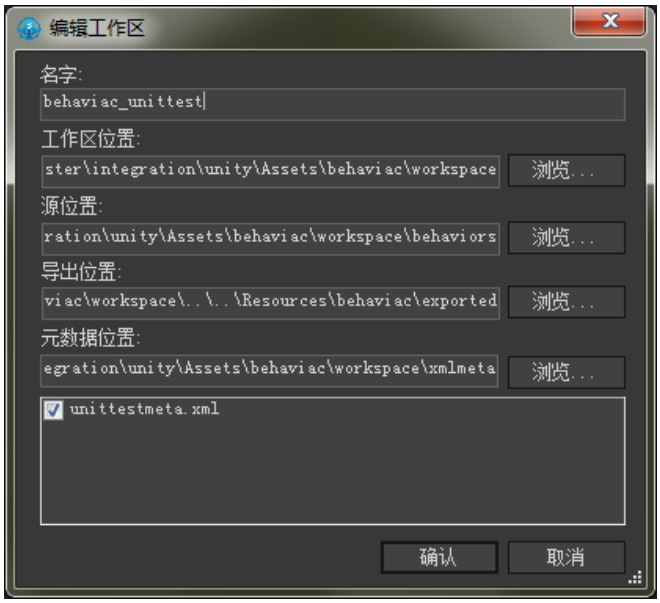

图 3.2.2 编辑工作区

行为树源文件和导出文件的区别在于，行为树源文件是在编辑器中供策划编辑使用的初始源文件（XML 格式），包含了很多冗余的信息，比如 UI 显示所需的属性等。而导出文件是一种精简版的行为树文件，只是给运行时端使用的一种高效的执行文件。

源文件就像那些 Raw Data，必须导出后才能在运行时中使用，而导出文件就像那些处理过并在游戏中直接使用的 Game Data。工作区文件、源文件、元信息文件等在**游戏中都不再需要**，游戏中**只需要导出的文件**。

目前支持 XML、BSON、C++和 C#四种导出格式。其中，XML/BSON 主要用于开发阶段，C++/C#主要用于最后的发布。C++/C#的内存和性能明显要优于 XML/BSON 的格式，所以我们建议在最后的发布过程，使用 C++/C#的行为树导出文件。

3.3 行为树

behaviac 组件提供了数量繁多、功能齐全的节点和附件，并支持扩展自己所需的节点类

型。此外，behaviac 组件还支持子树、事件、预制等高级用法，下面将一一进行介绍。

3.3.1 节点与附件

前面已经介绍，行为树的节点主要分为五大类节点或附件：动作、条件、组合、修饰、附件等，如图 3.3.1 所示。其中，动作、条件节点为叶子节点，组合、修饰节点为中间节点，附件必须附属在这四类节点上面而不能独立存在。

behaviac 组件还支持扩展新的节点类型，具体用法可以详见使用手册中的介绍。

图 3.3.1.1 行为树的节点和附件

为了支持子树（Subtree）的功能，behaviac 组件还提供了引用节点（ReferencedBehavior）类型，后文将对子树和引用节点详细介绍。

节点可以具有前置（Preaction）和后置（Postaction）等附件，如图 3.3.2 所示。前置可

以表示执行某个条件或者某个操作，如果是条件，那么前置必须返回成功后才能继续执行所在的节点，否则直接返回到父节点。而后置只用于表示该节点执行完毕后继续执行的操作。

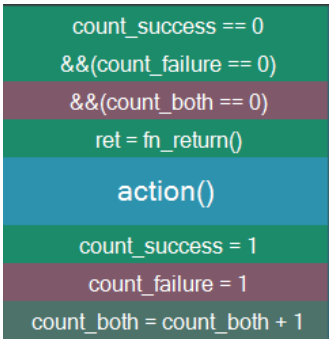

图 3.3.1.2 节点上的前置和后置

还有一种附件比较特殊，也即事件（Event）。事件是将一棵行为树作为另一棵行为树中某个节点（一般为根节点）的附件，用于响应事件发生时行为树的切换。后文将对事件进行更详细的说明。

为了支持某些特殊应用，behaviac 组件还为序列（Sequence）、选择（Selector）等组合节点添加了中断条件（Interrupted Condition），如图 3.3.1.3 所示。增加中断条件是为了让序列节点（或其他组合节点）在依次执行每个子节点的时候，都需要根据该中断条件的执行结果，来判断是否需要执行当前子节点。也即，所有的子节点有一个统一的中断条件被打断执行。

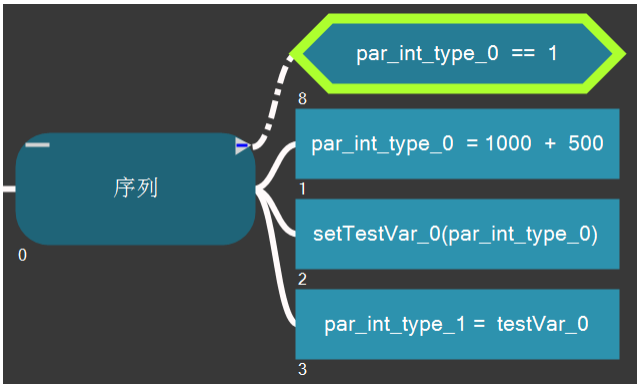

图 3.3.1.3 中断条件

在编辑器中，可以通过鼠标拖拽这些节点到新建的行为树中，或拖拽前置和后置附件到某个节点上。

3.3.2 预制

为了方便或加速行为树的编辑，behaviac 编辑器提供了预制（Prefab）的功能，以便利于复用已经编辑好的行为树。

behaviac 组件的预制跟 Unity 引擎中的预制工作机制类似，如果在用到预制的地方改变了预制中某个节点的属性，那么就会在所在的行为树中创建一份该预制的实例（Instance）。如果没有改变预制中任何节点的属性，则直接跟预制母体的属性保持一致，也即修改了预制

中任何节点的属性，用到该预制的其他行为树也跟着同步改变。

在编辑器中，可以将已经编辑好的行为树另存为预制，这些预制会组织在 Prefabs 目录下，以便区分于项目所需的行为树（归类在 Behaviors 目录下），如图 3.3.2.1 所示。

如图 3.3.2.2 所示，先构建了一个预制行为树“Prefab_安全的随机移动”。如图 3.3.2.3 所示，再将该行为树拖拽进行为树 Tank_SafeWander_RandomFire 中，可以看到该预制行为树会得到展开并复制过去，这加速了行为树的编辑，提高了开发工作的效率。

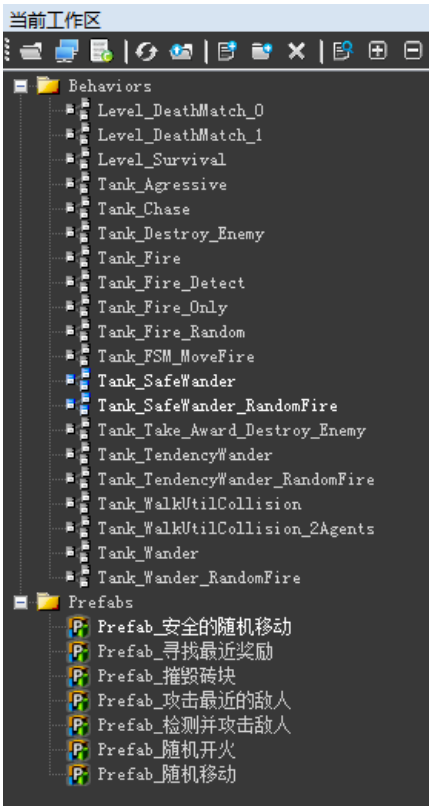

图 3.3.2.1 行为树与预制

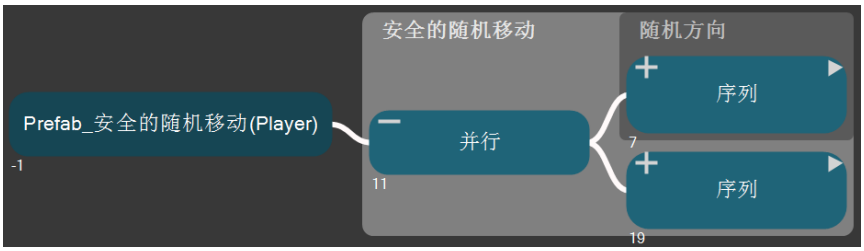

图 3.3.2.2 预制

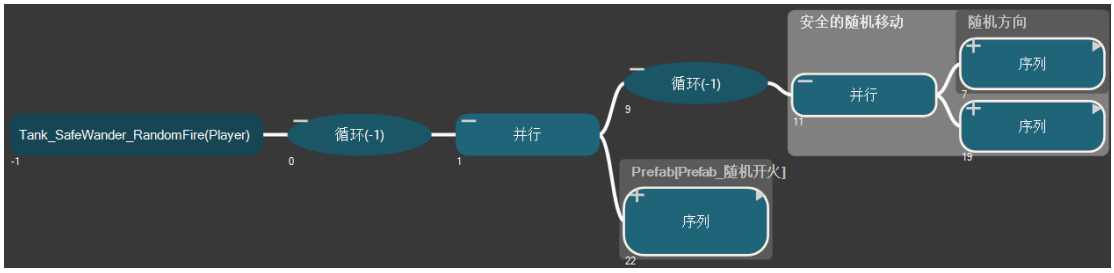

图 3.3.2.3 使用预制的行为树

https://github.com/TencentOpen/behaviac

3.3.3 子树与递归

在编辑器中，可以通过鼠标拖拽一棵行为树到另一棵行为树中，并作为引用节点（ReferencedBehavior）添加到另一棵行为树上。如图 3.3.3.1 所示，action_ut_0 就是 action_ut_1 的子树，action_ut_0 所在的节点是一个引用节点。

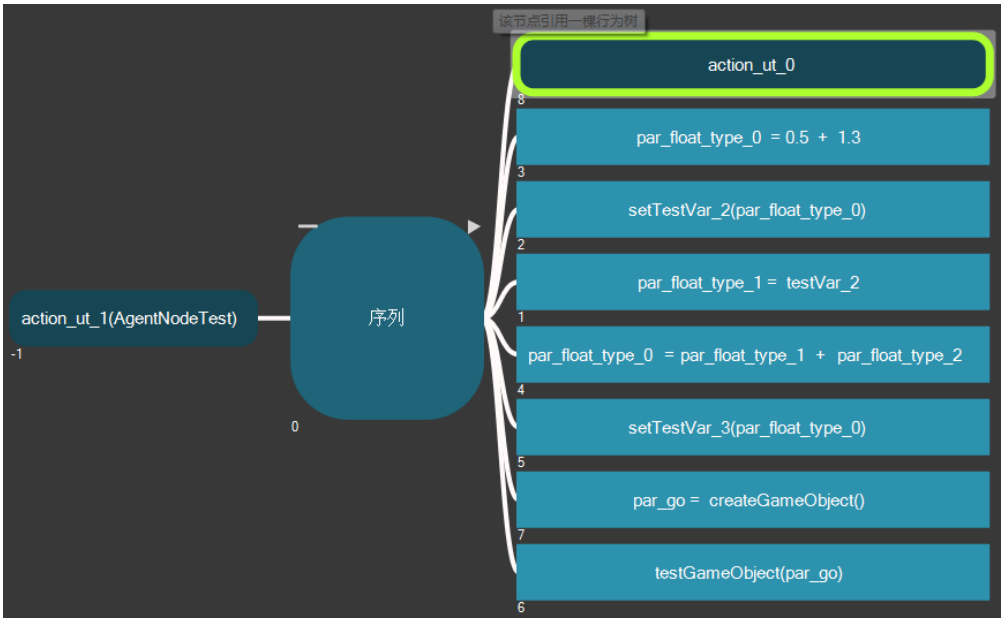

图 3.3.3.1 子树

有了子树的功能，behaviac 组件自然也支持了“递归”的功能，只要将当前行为树作为自己的子树即可。形成递归的时候需要注意不要造成死循环，这可以通过变量的使用来避免。

如图 3.3.3.2 所示，利用 testVar_0 来避免死循环：第一次进入的时候 testVar_0 == 0，所以可以执行下面的序列，先把 testVar_0 赋值为 1，那么在下面的递归重入的时候由于 testVar_0 == 1，所以 testVar_0 == 0 的条件不满足，所以下面的序列不会进入从而避免了死循环。

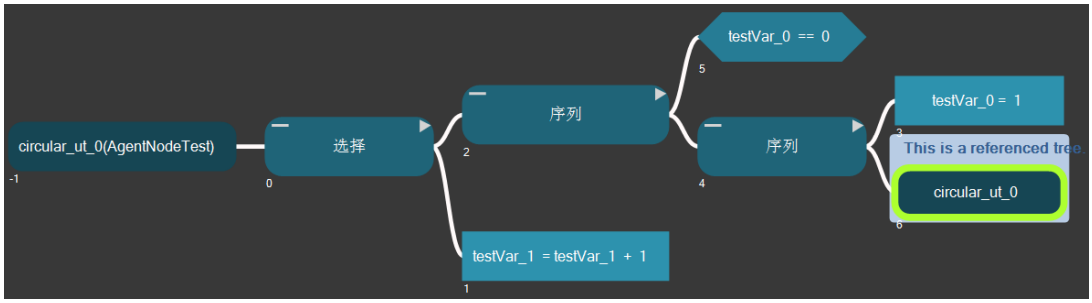

图 3.3.3.2 递归

3.3.4 事件处理

执行行为树的过程中，当状态、条件发生变化或发生事件（Event）时如何响应或打断当前的执行是个重要的问题。

目前 behaviac 组件支持三种方式来处理状态变化或事件发生：并行节点、选择监测节点、事件附件等。简而言之，并行和选择监测节点的工作方式是采用“轮询”的方式，每次执行时需要重新评估所有子节点，而不是像其他节点会保留上一次正在执行的子节点以便在下一次执行时继续执行。事件附件是在游戏逻辑发出事件时，才按需得到响应。

3.3.4.1 并行节点

依靠并行（Parallel）节点处理事件，需要把事件用条件的形式表达并且需要监控该条件，当该条件不满足的时候就退出。这种方式在概念上不太清晰，使用起来也比较繁琐。

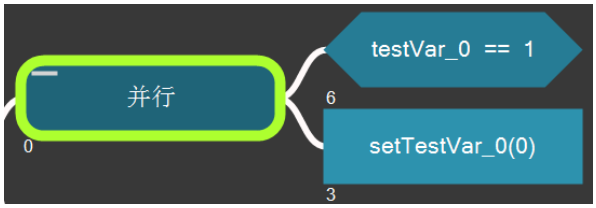

图 3.3.4.1 并行节点

3.3.4.2 选择监测节点

选择监测（SelectorLoop）和监测分支（WithPrecondition）节点作为对传统行为树的扩展，可以很自然的处理事件和状态的改变。选择监测和监测分支节点只能配对使用，即选择监测只能添加监测分支作为它的子节点，监测分支也只能作为选择监测的子节点被添加。

- 选择监测节点是一个动态的选择节点，和选择（Selector）节点相同的是，它选择第一个返回成功的子节点，但不同的是，它不是只选择一次，而是每次执行时都对其子节点重新进行选择。
- 监测分支节点有条件分支子树和动作分支子树。只有条件分支子树返回成功的时候，动作分支子树才能够被执行。

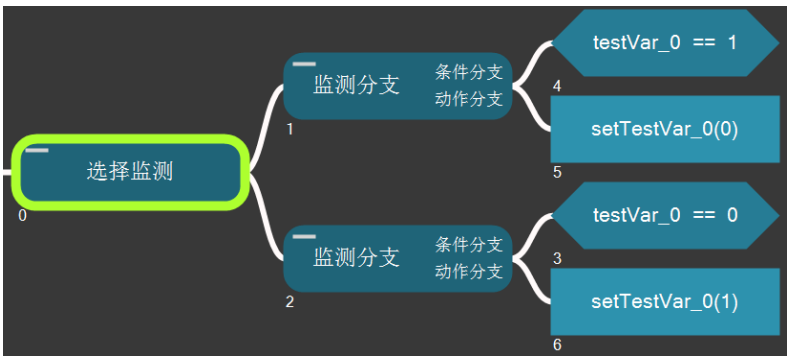

图 3.3.4.2 选择监测节点

3.3.4.3 事件附件

事件（Event）作为一种附件，是 behaviac 组件的一大特色。事件主要用于在游戏逻辑发出事件时，得到响应后打断当前正在执行的行为树，并切换到所设置的另一个行为树。

用一个具体例子来说明事件的原理和用法：

首先，在编辑器中为 AgentNodeTest 类添加了 event_test_int_bool(int val_int, bool val_bool)这个任务（Task）或者更形象的称之为“接口”（Interface），如图 3.3.4.3.1 所示。

其次，为行为树 event_subtree_2 添加一个任务节点，并作为根节点的第一个子节点，为该任务节点选择一个任务，这里我们直接选择 event_test_int_bool，如图 3.3.4.3.2 所示。类似于编程语言中的函数参数为函数体的代码提供了局部变量，任务节点中的接口参数为当前的行为树提供了局部变量，这些局部变量可以根据需要用于该行为树所有子节点。

再次，将上面的行为树 event_subtree_2 通过鼠标拖拽到行为树 event_ut_1 中的第一个序列节点上，这样该序列节点就有了一个事件的附件，如图 3.3.4.3.3 所示。

然后，为该事件设置参数，如图 3.3.4.3.3 所示。其中，“触发一次”表示该事件是否只触发一次就不再起作用。“触发模式”控制该事件触发后对当前行为树的影响以及被触发的子树结束时应该如何恢复，有转移（Transfer）和返回（Return）两个选项：

- **转移：**当子树结束时，当前行为树被中断和重置，该子树将被设置为当前行为树。
- **返回：**当子树结束时，返回控制到之前打断的地方继续执行。当前行为树直接“压”到执行堆栈上而不被中断和重置，该子树被设置为当前行为树，当该子树结束时，原本的那棵行为树从执行堆栈上“弹出”，并从当初的节点恢复执行。

最后，在游戏代码端通过如下代码，将事件“event_test_int_bool”发出，并制定所需的参数（这里是 15 和 true 两个值）：

```
testAgent.FireEvent("event_test_int_bool", 15, true);
```

这样，在执行行为树 event_ut_1 时，如果接收到事件“event_test_int_bool”，那么行为树中的事件附件将得到响应和处理，行为树的执行就会从当前的 event_ut_1 跳转到 event_subtree_2。

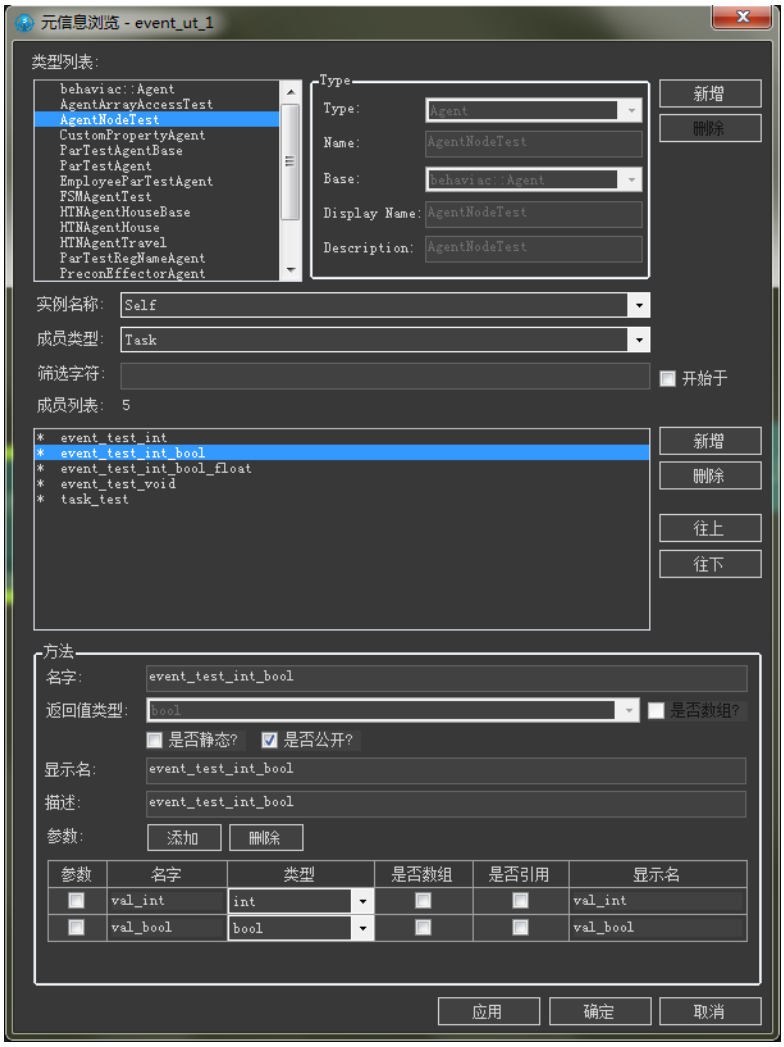

图 3.3.4.3.1 为 Agent 类创建一个任务

图 3.3.4.3.2 为行为树指定任务

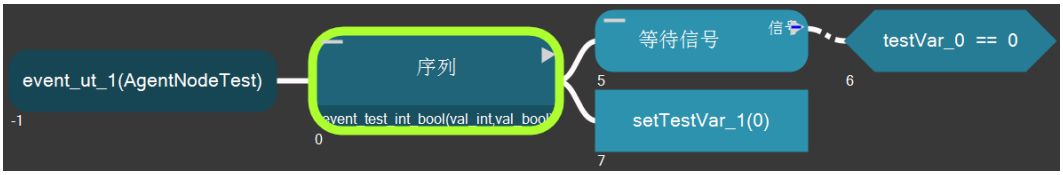

图 3.3.4.3.3 为行为树添加事件

图 3.3.4.3.4 为事件设置参数

另外需要补充说明的是，图 3.3.8 所表示的行为树 event_subtree_2 带有任务节点，我们也可以直接将该行为树拖拽到行为树 event_ut_1 中，如图 3.3.4.3.5 所示。这样在行为树 event_ut_1 中，选中引用节点 event_test_int_bool 后，就可以直接配置该子树执行时所需的参数（这里是 val_int 和 val_bool），如图 3.3.4.3.6 所示。

行为树的任务及其参数可以类比编程语言中的函数及其参数，因此 event_test_int_bool 这个“函数”有两个“形式参数”val_int 和 val_bool，而图 3.3.4.3.6 中所选择的 5 和 true 值就是 event_test_int_bool 函数执行时所用到的“实际参数”。

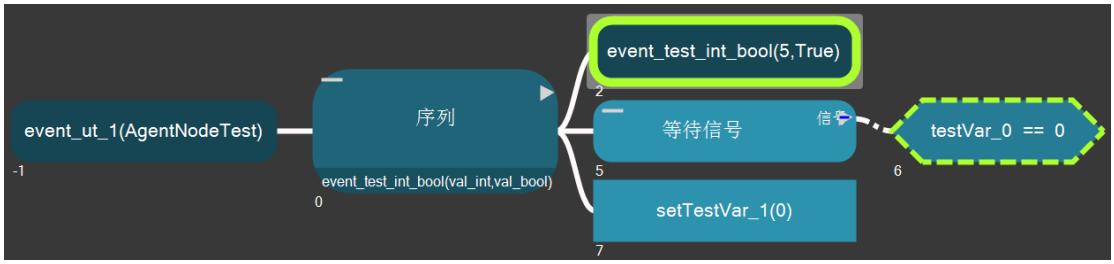

图 3.3.4.3.5 任务直接作为子树

图 3.3.4.3.6 为子树设置任务参数

3.3.5 热加载

behaviac 组件中的编辑器和运行时库都支持热加载，但是只针对 XML/BSON 格式的行为树文件。

在编辑器中，只要当前打开的行为树文件在编辑器外由于某种原因得到修改（例如通过项目中的文件版本管理系统强制同步行为树文件，或者通过文本编辑器强制修改行为树 XML 文件等），那么都可以自动的在编辑器中得到刷新。

对于运行时端（或游戏端），只要在编辑器中修改了行为树文件并重新导出，那么在游

戏运行过程中不用退出游戏，最新导出的行为树可以自动进行加载，这样就可以及时查看行为树最新的效果。

3.4 有限状态机

除了上面提及的行为树，behaviac 组件还支持传统的有限状态机（FSM）。在前面的 2.1 章节已经说明，有限状态机主要由状态（State）和转换（Transition）构成，如图 3.4.1 所示。

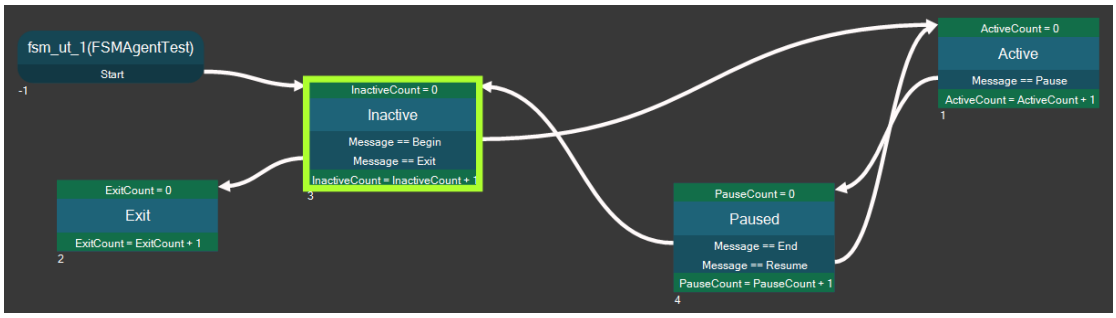

图 3.4.1 有限状态机

类似于前面提及的行为树子树的概念，behaviac 组件的一大特色就是支持将编辑好的一个有限状态机整体作为另一个有限状态机的“子树”，甚至有限状态机还可以跟行为树交替引用，互相作为彼此的子树，也即一个有限状态机可以整体作为一棵行为树的“引用节点”，同时一棵行为树也可以整体作为一个有限状态机中的“引用状态”。

在 behaviac 编辑器中，状态是作为类似行为树的节点一样独立存在，而转换是为类似行为树中的附件一样挂在某个状态上。有限状态机的编辑也很简单，只需要通过鼠标拖拽“状态”节点和“转换”附件到新建的有限状态机中，如图 3.4.2 所示。

图 3.4.2 有限状态机的状态和转换

类似于行为树中的节点，还可以为有限状态机中的状态节点添加前置和后置附件，以表示执行到当前状态节点时，所需要执行的前置条件和后置操作。

3.5 调试

编辑器支持跟游戏端进行连调，如图 3.5.1 所示。支持查看属性的变化（图 3.5.2）、在节点上设置断点（图 3.5.3）、高亮显示行为树的执行路径（图 3.5.3）以及记录每帧信息以便后续查看等功能。

图 3.5.1 连接游戏

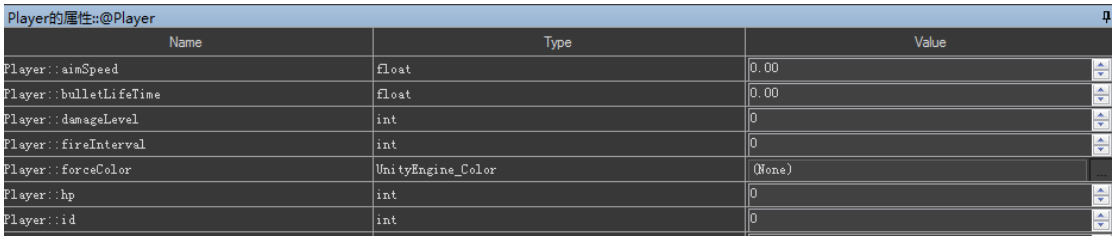

图 3.5.2 查看 Agent 的属性

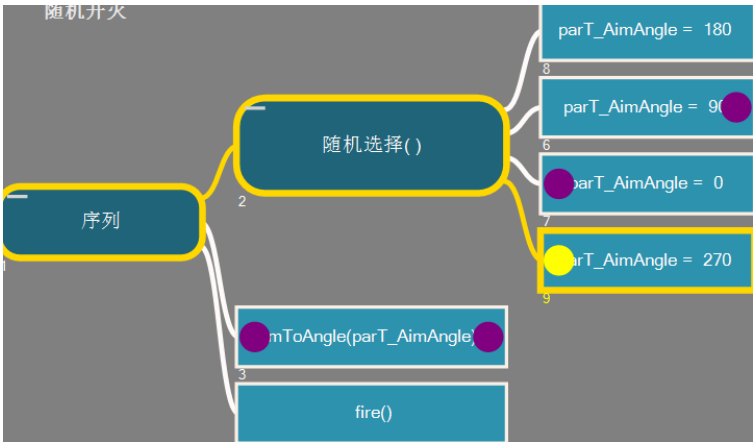

图 3.5.3 设置断点以及高亮显示执行路径

3.6 单元测试与 Demo

为了方便入门，behaviac 组件提供了 C++和 C#两种编程语言的单元测试以及游戏 Demo，可以通过编辑器中的菜单项“帮助”->“控制说明” 进入，然后点击快速打开所需的单元测试或游戏 Demo 所用到的工作区，如图 3.6.1 所示。

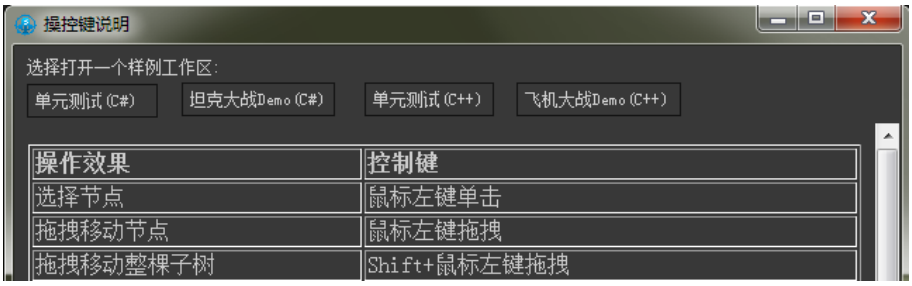

图 3.6.1 单元测试与 Demo

3.7 开源

behaviac 组件已在 github 上开源，网址是 https://github.com/TencentOpen/behaviac，可以下载最新的安装包、源码以及相关文档等。

另外，behaviac 视频讲解的链接为：http://gad.qq.com/tool/detail/2